AD-A094 653 ARMSTRONG (PHILIP N) SANTA ANA CA F/G 12/1
SPARSE MATRIX COMPUTATIONS WITH A SPECIAL ARRAY-PROCESSOR.(U)
DEC 80 P N ARMSTRONG, D G CANTOR N00014-79-C-0477

UNCLASSIFIED NL

1 OF 1
AD
A094653

END
DATE
FILMED
3 81
DTIC

FINAL TECHNICAL REPORT ON
SPARSE MATRIX COMPUTATIONS WITH A SPECIAL ARRAY-PROCESSOR

Sponsored by

Office of Naval Research

Contract N00014-79-C-0477

Contractor:

Philip N. Armstrong

17331 Keegan Way

Santa Ana, CA 92705

December 1980

FINAL REPORT

This is the final report of a study of special computing machinery for computations on sparse matrices. under contract with the Office of Naval Research N00014-79-C-0477

The elements of the computing system and its application to Gaussian elimination are described in some detail in the paper "On a Special Purpose Matrix Array-Processor" by Philip N. Armstrong and David G. Cantor, which has been submitted for publication and comprises the major part of this report. This paper describes the system and the algorithms for which it is designed, so that a reader who is acquainted with computations on matrices will understand the system's utility for such computations. Some additional remarks regarding selected matrix computations may be useful and are included here, although what is written below may be inferred from the substance of the paper. Reading the paper is a pre-requisite for these remarks.

1. Matrix transformations of a vector: computing Ax and A^Tx.

Assume that A is an $n \times n$ matrix and that column a_{*j} of A is stored in module M_j, i.e. A is treated as though it were dense. If the j^{th} element of x is in processor P_j, module M_j, computation of either Ax or A^Tx will require n ticks. More generally, if A has m rows and n columns, stored in the above manner, $\max(m,n)$ ticks will be required.

2. Forward and back substitution: $y = L^{-1}b$ and $x = U^{-1}y$.

Suppose L is an $n \times n$ lower triangular matrix, U is an upper triangular matrix, and b is a vector with n components. The computations $L^{-1}b$ and $U^{-1}y$ will each require n ticks. The n^2 entries of L must be accessible in order of increasing row indices; the entires of U must be accessible in order of decreasing row indices.

3. Solving the system $Ax = b$, where A is an $n \times n$ invertible matrix.

The entries of x may be obtained by Gaussian elimination with partial pivoting in $n^2 + 3n$ ticks, assuming fill-in does not exceed the memory capacity of the system. After the first n^2 ticks, the vector Pb may be stored, where P is the permutation matrix defined by the partial pivoting operations, so that in $n^2 + n$ ticks the LU form will be completed and the n components of Pb will be in place. The systems $Ly = Pb$ and $Ux = y$ may then be solved in the following $2n$ ticks. This amount of time may be decreased if A is sparse. For details, see page 20 of the paper.

4. Updating the LU form of an $n \times n$ matrix.

Suppose the LU form of a matrix has been obtained by Gaussian elimination with partial pivoting, using the array-processor. Then suppose the matrix A is changed to a new matrix $\bar{A}$ by the exchange of a column a_{*j} for a new column $\bar{a}_{*j}$. This is the "pivot step" of the Simplex algorithm for linear programming, as described in the paper "Numerical Techniques in Mathematical Programming" by R.H. Bartels, G.H. Golub, and M.A. Saunders, in the book Nonlinear Programming, Academic Press, New York, 1970. This computation requires no more than $5n - j + 1$ ticks, using the method described in the paper. In this method, an upper Hessenberg matrix is reduced to its LU form and the elements of the lower triangular matrix which is produced are stored, using no more storage than that allocated for one column of the original matrix A. Similar computations will produce matrices factored in this way with new column interchanges. With each subsequent decomposition, the storage will be similarly increased, and the amount of time consumed by the procedure will increase by one tick. Periodically, recomputation of the LU form will be performed (as is customary in applications of the Simplex algorithm).

5. Householder transformations.

If A is an $m \times n$ matrix stored by column in the modules so that each column uses one module of the array, A may be reduced to a matrix R by an orthogonal matrix Q, which is the product of Householder transformations of the form $I - \beta_k u_k u_k^T$. Suppose A_k is produced by k such transformations, starting with matrix A $(= A_0)$. The first step in computing A_k is to scan the last $n - k + 1$ columns of A_{k-1} and compute the lengths of these columns. The column of greatest length will be selected and moved to the k^{th} column position. This selection and movement of the column will consume $2n + j - k$ ticks, where a_{*j} is the selected column. There will then remain $2(n - k)$ ticks for the transformation itself. When the entries of several columns are in a single module, the selection of the column of greatest length may be done by scanning each column twice so that the columns may be sorted by column length, after which the requisite column permutation and arithmetic computations may be done. This selection procedure illustrates the utility of the flag bits and the facilities within the processors for changing the keys over which the words are sorted.

6. Permutations of rows and columns.

Permutations on the rows and columns of matrixes to reduce fill-in or to increase numerical stability are discussed in the literature.

It suffices to note that all standard permutations can be performed. For example, it is often convenient to arrange the columns of a matrix so that the column indices of the first non-zero row entries are in increasing order. This requires a perliminary scan of the matrix, of course. Other procedures, e.g. formation of the transitive closure of the graph represented by the non-zero entries of the matrix and determining whether this matrix is symmetric are easily performed in this system.

7. Computations which require augmentation of the system.

As may be seen from (1) of this report, no extra time is required for forming the transpose of a matrix when computing $A^T x$ instead of Ax. There are, however, computations such as the one above in determining the symmetry of the transitive closure, or two-dimensional Fourier transforms, for which the transpose of a matrix is either required, or at least useful. Thus, it may be desirable to provide data paths to expedite this procedure. If module M_i is connected to those modules M_j for which $i - j$ is a power of 2, an additional (approximately) $\log_2 n$ data paths will be connected to each mocule, where n is the total number of modules in the array. This provision makes the computation of the two-dimensional Fourier transform on n^2 entries in approximately $2n \log_2 n$ ticks feasible -- of there are at least n modules in the system.

It appears that many other computations may be made feasible, or expedited, if more than one SSM is provided at each module. An example of this use of storage is given on page 20 of the paper. This matter remains to be examined, along with the improvement in system performance to be expected from additional memory elements.

Philip N. Armstrong

December 15, 1980

On a special purpose matrix array-processor

Philip N. Armstrong[1] and David G. Cantor[2]

The advent of very large scale integrated circuitry (VLSI) has increased the interest in special purpose array-processors, which perform certain algorithms extremely rapidly, for use as peripheral devices on computers. To be economical, these processors should be built from many copies of a few simple devices, with interconnections kept to a minimum. They should allow for parallel processing and pipelining. See, for example, Jones and Schwartz (and the references cited therein) [5].

The purpose of this paper is to present a novel design for a matrix array-processor. Such designs have been presented before. See for example "Systolic arrays (for VLSI)" by H.T. Kung and Charles E. Leiserson [7] or Chapter 8 of Mead and Conway [8]. For a general survey of parallel matrix computation schemes, see Heller (and the references cited therein) [3].

Our proposed system is radically simpler, and correspondingly slower, than that of Kung and Leiserson. For example to invert a dense n by n matrix, their system would require n^2 processors and take n basic units of time, while ours has no particular requirement on the number of processors, but with n (or more) processors takes n^2 units of time.

However, our system requires far fewer interconnections, thus allowing its extension onto numerous VLSI chips (as and if necessary). In addition it is able to perform all standard matrix operations including partial pivoting during Gaussian

1. Supported in part by ONR Contract N0014-79-C-0477

2. Supported in part by NSF Grant MCS-79-03711

elimination as well as updating of the LU form of a basis, which is required, for example, during use of the (revised form of the) simplex algorithm (see Orchard-Hays [9] and Tomlin [12]). The proposed system can efficiently perform both Givens and Householder transformations in parallel [4], preliminary to computation of eigenvalues and eigenvectors of (not necessarily symmetric) matrices. It takes advantage of sparse matrices, even if irregular (i.e. not necessarily band-limited matrices). Finally, our system can efficiently perform the Fast Fourier Transform and provides an excellent sorting mechanism for lists of numbers. We give examples of how some of the above are performed.

The major novelty of our system is a non-standard memory, not addressable (in the usual sense) and thus not requiring the extensive addressing logic (and associated connections) used in ordinary computer memory, together with appropriate formulations of the standard algorithms which work with this type of memory. The memory provides a type of automatic list-processing.

1. The memory

The memory consists of independent modules, each a "self-sorting" memory (SSM), as described by Armstrong and Rem in [1]. A simplified version of the memory is illustrated in figure 1. It consists of two sequences of binary shift registers $U_1, U_2, \ldots, U_N$ and $L_1, L_2, \ldots, L_N$. All of the U_i hold $u \geq 1$ bits and all of L_i hold $\ell \geq 1$ bits; $w = u + \ell$ is called the <u>word length</u> of the memory (typical values for w range from 50 to 200). The U_i and L_i are interconnected by logical units C_i called <u>sorters</u> [6]. A sorter can be in one of three states: U (undetermined), S (straight-through) or N (normal). There are (simple) controls on the memory to place each of its sorters into the state U at a specified time.

The sorter has two inputs: the left one is called the <u>low input</u> and the right one is called the <u>high input</u>. Similarly there are two outputs: the left one is called the <u>low output</u> and the right one is called the <u>high input</u>. In state N the low input is connected to the low output and the high input is connected tc the high output. In state S, this is reversed, and the low input is connected to the high output and the high input is connected to the low output. See figure 2.

The shift registers all operate in synchronism, shifting one bit each <u>bit time</u>. If a sorter is in state U and the two inputs are the same it will remain in state U and transmit (without delay) the (common) value of the inputs to both outputs. If however the low input receives a 0 and the high input receives a 1, then the sorter goes into state N, while if the low input receives a 1 and the high input receives a 0 the sorter goes into state S. Once in state N or S, the sorter remains in that state until a control signal changes it.

The upper left input of the entire module is (similarly) called the module's <u>low input</u> and the lower right input is called the <u>high input</u>. Similarly the lower left and upper right outputs are called, respectively, the <u>low output</u> and the <u>high output</u>.

The period of this memory is w (bit times). We assign a start time of $s_j = ju$ (reduced modulo w) to the sorter C_j, $0 \leq j \leq N$.

During normal operation, the controls put sorter C_j into state U at the beginning of bit time s_j. (This is done for all sorters, once each period. For the customary values of u (1, $w/2$, or $w-1$), implementation is extremely simple.

The registers U_j and L_j together form a ring R_j. Each ring R_j holds w bits.

We shall say that the sorter is stable if (1) during one period all sorters stay in state U or go into state N; (2) zeros are fed into the low input and ones into the high input of the module; (3) sorter C_j is set to state U at time s_j. When that is so, each ring R_j contains w bits. The bit entering C_j at time s_j (equivalently C_{j-1} at time s_{j-1}) is called bit b_0 (or b_{0j} if we wish to identify the ring). Successive bits, proceeding counter-clockwise around the ring, are called bits $b_1, b_2, \ldots, b_{w-1}$. Thus the w bits in ring R_j represent a binary number $b_0 b_2 \cdots b_{w-1}$ ($= \sum_{i=0}^{w-1} b_i 2^{w-1-i}$ in decimal); b_0 is the high-order (most significant) bit, b_1 is the next most significant bit, ... , and finally b_{w-1} is the low order (least significant bit). Denote the word $b_{0j} b_{1j} \cdots b_{w-1,j}$ by B_j; it is the word cycling in the j^{th} ring. If the words B_j and B_{j+1} are equal then sorter C_j will remain in state U. If they are not equal, then there must be a first bit, say the k^{th}, in which they differ. Then $b_{i,j-1} = b_{i,j}$ for $0 \leq i \leq k-1$ but $b_{k,j-1} \neq b_{k,j}$. By our assumption of stability, when these bits are encountered C_j goes into state N. This implies that $b_{k,j-1} = 0$ and $b_{k,j} = 1$. It follows that $B_{j-1} < B_j$. Thus we see that during stable operation, the words B_j satisfy $B_1 \leq B_2 \leq \cdots \leq B_N$.

Let us see what happens if a word $B_0 = b_{00}b_{10}\cdots b_{w-1,0}$ is entered into the low input, replacing the zeros customarily entering there, with b_{00} entering at time s_0. If $B_0 \leq B_1$ it is clear that B_0 will be output immediately from the low output. If however $B_0 > B_1$, then B_1 will be output immediately and B_0 will be sent to C_2, where it will be compared with B_2. If $B_0 \leq B_2$ then B_0 will remain in ring R_1, otherwise it will be sent to C_3 and compared to B_3. This will proceed until a stable situation is reached. At this time the memory will contain numbers $B_1' \leq B_2' \leq \cdots \leq B_N'$. Thus we observe the basic operation of this memory, which is to insert a number B_0 into the low input and obtain from the low output the smallest number among $B_0, B_1, \ldots, B_{N-1}$. Since this is a "pipe-lined" memory, the basic operation can be repeated every period, even though the memory is not stable.

Similarly if a number B_{N+1} is entered into the high input, the larger number among B_N, B_{N+1} will be output from the high output, and the remaining numbers will move into increasing order.

Note that while one can successively enter numbers from the low input, once each period, or from the high input, once each period, in order to switch from one input to the other, it is necessary for the "self-sorting" memory to stabilize, and this may take as many as N periods.

In order to avoid this lengthy delay, we use the slightly more complicated memory shown in figure 3. The elements $D_0, D_1, \ldots, D_{N-1}$ are sorters, while the elements $V_1, V_2, \cdots, V_{N-1}$ are shift registers, each holding u bits. The sorters D_j are drawn upside down (compared with the C_j); thus the lower left input is the low input and the upper left output is the low output. Similarly the other input and output are the high input and output, respectively. Like C_j, D_j is put into state U at time s_j.

During stable operation, the U_j, L_j and C_j function as before. Ring R_j contains word W_j and $W_1 \leq W_2 \leq \cdots \leq W_N$. The V_i contain ones. If a number W_0 is entered into the low input the system behaves as before. If a number W_0 is entered into the high input, then it will travel along the path $V_1, V_2, \ldots, V_{N-1}$. If any of the numbers in the rings $R_1, R_2, \ldots, R_N$ is greater than W_0, it will be compared with W_0 at some stage and replace W_0. Thus $(N-1)u$ bit times after the insertion of W_0, the largest of the numbers $W_0, W_1, \ldots, W_N$ will emerge from the high output (for our purposes, this delay is unimportant).

Finally we will need one more control. This will connect to all sorters C_j and D_j. It will enable, for given k, replacement of the k^{th} bit of each word, as it enters the sorter, by 0 (this will only be done for $2 \leq k \leq w$, and usually for $2 \leq k \leq 6$).

Before explaining how this self-sorting memory is used, we note that if, for purpuses of VLSI implementation, the connection from U_N to L_N is made external to the chip containing the module, and the two control signals (the first control signal puts sorter C_j and D_j into state U at time s_j; the second sets the k^{th} bit of each word of the memory to 0) are passed both in and out of the chip, self-sorting memories can be strung together, one after another, and chip limitations do not limit the size of the memory, nor do time delays arising from the finite speed of pulse transmission). In addition, there are rel ively simple modifications, which allow what is, in essence, bit parallel operation of these memories, and which for a given clock speed, allow a much greater memory speed. We do not describe these here, although they might be used in an actual implementation.

2. The array-processor

The array-processor consists of a sequence of M identical modules as shown in figure 4. Each module consists of a processor (P) and attached self-sorting memory holding N words. Each SSM connects only to its processor, and its lowest ring is an addressable register in its processor. Each processor, except those at the end, are connected by bi-directional lines to each of the eight adjacent processors (four on each side). For simplicity these connections are not shown on the diagram. In addition, there are two common buses to which all processors have access. Attached to one of these buses is one reciprocal unit (which given $x \neq 0$, returns $1/x$) and one square-root unit (which given $x \geq 0$ returns $\sqrt{x}$). The square root unit and reciprocal unit may be physically the same. The processors are numbered $P_1, P_2, \ldots, P_M$.

The processors have limited capability. They have a few registers for holding floating point numbers. They can add and multiply both fixed and floating point numbers (division and square root are not necessary), compare floating point numbers, and perform limited bit-manipulation operations. They have a small program-store (which can be in ROM, or in RAM which is down-loaded from the controlling computer). It is convenient (for the moment) to assume the processors operate in a synchronous mode. We shall call the time necessary to perform one period of the self-sorting module (w bit times) followed by a floating point operation of the complexity $a,b,c \rightarrow a \mp b \cdot c$ (or similar operation) a tick. We assume that during the first part of the tick (when the processor is communicating with its self-sorting memory) a processor can also send or receive (up to three) words of w bits from its adjacent processors (i.e., interprocessor communication time is no more than one self-sorting module period). Later we shall show that these restrictions are not all necessary, and that the array-processor can take advantage of faster inter-processor communication (especially when working with sparse matrices). It is convenient to assume that the reciprocal unit and the square-root unit also function (including communication) in one tick. However, if this is not so, the entire processor will function only slightly slower. For example, in Gaussian elimination of an n by n matrix, n divisions are required, and it is the extra time for these n divisions that would have to be included.

As we shall see, an arbitrary (invertible) dense $n \times n$ matrix can be inverted in n^2 ticks (assuming our array contains at least $n + 2$ processors). Thus if a tick takes 10 μs, then a 500 by 500 dense matrix can be inverted in 2.5 seconds. We should also note, in this regard, that the primary limitation, on the size of matrices that can be handled, is that the total memory capacity (of the self-sorting memories) suffices to hold the original matrix and all intervening matrices occurring during the computation (in general, each self-sorting memory will hold 1 or more columns of a matrix, but a column must be entirely contained within one self-sorting memory).

3. Data manipulation

We shall assume throughout the remainder of this paper that all data items have a leading 0, i.e. $b_0 = 0$. We shall call any word with a leading 1 ($b_0 = 1$) a blank. To the self-sorting memory (SSM), blanks are larger than any valid datum, hence, during computations, each SSM will contain valid data followed by blanks.

Note that a processor can perform the following three operations on its attached SSM.

1) it can delete the least item (which will be transferred into a register of that processor) by inserting a blank into the low input of the SSM;

2) it can add an item into the SSM (displacing a blank if any are present, otherwise displacing the largest datum, which will be lost) by inserting the item into the high input;

3) it can enter a new item into the low input; the SSM will simultaneously write out the smallest of the valid data items (including the new one) present.

Let us first see how the array-processor sorts a list of $L \leq MN$ numbers in $2L + N$ ticks. We first (simultaneously) load all of the SSM with blanks. This will take N ticks using step (1) above (with a slightly more elaborate SSM control, this could be done in 1 tick).

The data items are sent one-by-one into SSM 1, using step (2) until N items have been entered. Then it will continue inserting items using step(3), the displaced items will be sent by SSM 2, where they will be inserted as before, then to SSM 3, etc., until SSM k $(k \leq M)$ is partially filled. Then, starting with SSM k, they will be written out, one-by-one, using step (1). The items will then appear in ascending sorted order. If the items to be sorted are keys k_i from larger records r_i, suppose p_i is a pointer to r_i and form the word $x_i = (k_i, p_i)$ (i.e. the pointer p_i is concatenated to the key k_i). As long as the total number of bits in k_i and p_i is less than w, the keys and pointers may be sorted together. If descending order of sort is desired, processor P_1 may be programmed to take the ones complement of the x_i as they enter and recomplement when they leave. Other preliminary transformations may be necessary if the k_i represent signed-magnitude or twos-complement numbers. Such transformations can also be performed in P_1.

We now show how a single processor-SSM combination can perform data manipulation on vectors. We describe a slightly simplified version of our procedure. We represent a vector entry as follows.

0	F	Index (I)	Value (V)

Here the leading 0 is that required of all valid data; F denotes a one bit "flag" field; I is the subscript or index of the given vector entry; finally V is its numerical value (probably in floating point). Note that the flag and index are the high order part of the word, not its value, and thus the SSM will sort on indices, not on the value (as we shall see, in Gaussian elimination, the pivot entry is determined by comparisons within the processor, not by the SSM). If $V = 0$, then the word is not stored in the SSM. This last rule enables us to take advantage of sparse vectors. In practice, I might be a 20 bit field and V an 80 bit field.

Let us suppose that an SSM contains all the (non-zero) elements of a vector, and that we wish to read them out one-by-one, modify them, and then replace them. We assume all the flag bits are 0 (there is a control pulse in the SSM to set them to 0, if necessary).

Using operation (1) we move the first element of the SSM into the processor. After we modify it, we set its flag bit to 1 and use operation (3) to replace it, and get the next element. Since the first element is large, it will "bubble" to the top of the SSM. We repeat this operation until we have passed through all valid data items. The latter will be signalled, either by a count, known in advance, or by the appearance of an element with flag bit 1. At that point, we complete the operation by setting all flag bits to 0. To interchange two elements, say the i^{th} and the j^{th}, we simply replace the index i by the index j in the i^{th} element, when it appears, and replace the index j by the index i when the j^{th} word appears.

We note that, in a similar fashion, the single processor-combination can perform the "perfect shuffle" of Stone [11], which is used in the Fast Fourier Transform (and numerous other algorithms). Recall that in order to perform the perfect shuffle, it is necessary to have $n = 2^k$ items, and that item j, $0 \leq j < n$, is sent to item $2j$ if $2j < n$ and to item $2j+1-n$ if $2j \geq n$ (this amounts to a circular left shift by 1 of the index). The flag bit is used as before.

In order to perform an FFT, all of the n^{th} roots of unity are needed, in an appropriate order. It is not hard to see that two (since they are complex) SSM's could be used to hold the appropriate roots of unity. Thus the array processor could perform $\lfloor K/2 \rfloor - 2$ simultaneous FFT's in $2kn$ ticks.

We note finally, that any permutation π of a vector for which $\pi(i)$ is an "easily computable" function of i can be performed in the SSM.

By using two adjacent processor-SSM's we can apply an arbitrary permutation τ to the indices of a vector. One SSM holds words with index field i and data field $\tau(i)$ (if $\tau(i) = i$, the word need not be stored). The vector entries and permutations are read out, and i is replaced by $\tau(i)$ in the vector entry, the flag bits are set to 1, and the process continues.

We note, finally, that many of the algorithms will require more than one flag bit, and we usually will reserve three flag bits.

4. Gaussian Elimination

The purpose of this section is to describe the algorithm for performing Gaussian elimination, with partial pivoting, on dense matrices, using the array-processor. While basically very simple, the algorithm because of its non-standard order of computation and parallel computations is somewhat difficult to describe. Let us recall the standard Gaussian elimination for transforming an n by n matrix A to LU form [4]. We wish to write $A = LU$ where L is a lower diagonal matrix with ones on the diagonal and U is an upper diagonal matrix. We define a sequence of matrices $A^{(k)} = (a_{ij}^{(k)})$, $0 \leq i,j,k \leq n$, with $A^{(0)} = A$ and then successively for $k = 1,2,\ldots,n$, $A^{(k)}$ is defined by

$$(1) \qquad a_{ij}^{(k)} = a_{ij}^{(k-1)}/a_{kk}^{(k)}, \qquad k+1 \leq i \leq n;$$

$$(2) \qquad a_{ij}^{(k)} = a_{ij}^{(k-1)} - a_{ik}^{(k-1)}a_{kj}^{(k-1)}/a_{kk}^{(k)}$$

$$= a_{ij}^{(k-1)} - a_{ik}^{(k)}a_{kj}^{(k-1)}, \qquad k+1 \leq i,j \leq n;$$

$$(3) \qquad a_{kj}^{(k)} = a_{kj}^{(k-1)} \qquad \text{otherwise}.$$

Then $A^{(n)} = (L - I) + U$; i.e. the elements below the main diagonal of $A^{(n)}$ equal the corresponding elements of L (of course the diagonal elements of L are 1), while the elements on or above the main diagonal equal the corresponding elements of U. The above algorithm somewhat rashly assumes that the "pivot elements" $a_{kk}^{(k)}$, $1 \leq k \leq n$, are not zero (and to avoid numerical instabilities not too "small"). We shall later drop this assumption, but continue it for the moment.

The normal implementation of Gaussian elimination is given by (using a self-explanatory "Pidgin" PL/1):

```
do  k = 1 to  n;
    do  i = k + 1  to  n;
        $a_{ik} = a_{ik}/a_{kk}$;
    end;
    do  i = k + 1  to  n;
        do  j = k + 1  to  n;
            $a_{ij} = a_{ij} - a_{ik} * a_{kj}$;
        end;
    end;
end;
```

It is easy to verify that the outer loop on k computes $A^{(k)}$ from $A^{(k-1)}$.

Let us denote the i^{th} row of a matrix $A = (a_{ij})$ by a_{i*} and the j^{th} column by a_{*j}. The Doolittle method of Gaussian elimination is obtained by noting that, in the above algorithm $a_{*k}^{(k)} = a_{*k}^{(k+1)} = \cdots = a_{*k}^{(n)}$, and computing $A^{(n)}$ column by column. This yields the following algorithm.

```
do  k = 1  to  n;
     do  j = 1  to  k-1;
          do  i = j + 1  to  n;
               $a_{ij} = a_{ij} - a_{ik} * a_{kj}$;
          end;
     end;
     do  i = k + 1  to  n;
          $a_{ik} = a_{ik}/a_{kk}$;
     end;
end;
```

It is a modified version of this latter algorithm that we shall implement. Note that the $n - j$ computations $a_{ij} = a_{ij} - a_{ik} * a_{kj}$ of the inner loop above may be performed simultaneously (and this is what, in essence, we shall do). The above algorithm can then be conveniently implemented, in a parallel fashion, on our array-processor as follows: We compute the columns successively. To compute the k^{th} column, the elements $a_{1k}^{(0)}, a_{2k}^{(0)}, \ldots, a_{nk}^{(0)}$ are fed in successively, one per tick, from the left. When element $a_{1k}^{(0)}$ $(= a_{1k}^{(n)})$ enters P_1 it is held in a special register. One tick later it is sent to P_2. During the same tick $a_{2k}^{(0)}$ enters P_1, $a_{21}^{(1)}$ is brought out of SSM_1 and $a_{2k}^{(n)} = a_{2k}^{(0)} - a_{21}^{(1)} a_{1k}^{(1)}$ is computed in P_1. During the next tick the following operations are performed:

(i) $a_{1k}^{(k)}$ moves from P_2 to P_3,

(ii) $a_{2k}^{(k)}$ moves from P_1 to P_2, where it is held for later use, and

(iii) $a_{3k}^{(1)} = a_{3k}^{(0)} - a_{31}^{(1)} a_{1k}^{(1)}$ is computed in P_1. This continues through the first $k - 1$ columns. As the elements

$a_{ik}^{(k)}$, $i \leq k$ enter P_k they are passed into SSM_k. When $a_{kk}^{(k)}$ enters, it is passed into SSM_k, and its reciprocal is calculated in the reciprocal unit and saved in P_k. When the elements $a_{ik}^{(k-1)}$, $i < k \leq n$, enter P_k, the element $a_{ik}^{(n)} = a_{ik}^{(k)} = a_{ik}^{(k-1)} \cdot (1/a_{kk}^{(k)})$ is computed and sent to SSM_k. Note that in the tick following computation of $a_{nk}^{(1)} = a_{nk}^{(0)} - a_{n1}^{(1)} a_{1k}^{(0)}$ in P_1, $a_{1,k+1}^{(0)}$ enters P_1 and the above sequence of computations is repeated (with the appropriate changes) for the (k+1)st column.

If the ticks are numbered so that $a_{11}^{(0)}$ enters C_1 at tick 1, then $a_{1k}^{(0)}$ enters P_1 at tick $n(k-1) + 1$ and $a_{1k}^{(k)}$ enters P_k at time $(n+1)(k-1) + 1$. At tick n^2, $a_{nn}^{(1)}$ is computed in P_1, and at time $n^2 + 1$ we may start outputting the first column of $A^{(n)}$. At tick $n^2 + n + 1$, all entries of $A^{(n)}$ have been computed.

Let us now investigate the changes necessary in order to allow partial pivoting. Recall that this means that when computing $A^{(k)}$ from $A^{(k-1)}$ the largest element in column k is chosen as a pivot element. Specifically, suppose that after $A^{(k-1)}$ has been chosen, p_k is chosen, $k \leq p_k \leq n$, so that $|a_{p_k k}^{(k-1)}| = \max_{k \leq i \leq n} |a_{ik}^{(k-1)}|$. Then a permutation τ_k of $1,2,\ldots,n$ is chosen which fixes $1,2,\ldots,k-1$, replaces k by p_k, and is otherwise arbitrary. Two convenient choices are $\tau_k = (k,p_k)$ (i.e. k and p_k are interchanged) and $\tau_k = (k,k+1,\ldots,p_k)$ (i.e. p_{k+1} replaces k, and $k,k+1,\ldots,k+p_k-1$ each replace their successors). Let σ_k be the permutation matrix which, by left multiplication, permutes the rows of $A^{(k-1)}$ according to the permutation τ_k. Then before computing $A^{(k)}$, $A^{(k-1)}$ is replaced by $\sigma_k A^{(k-1)}$ and computation proceeds as before. Put $S_k = \sigma_k\sigma_{k-1}\cdots\sigma_1$ and $T_k = \tau_k\tau_{k-1}\cdots\tau_1$. If we were omniscient and knew in advance the permutations $\pi_1,\pi_2,\ldots,\pi_n$, we could apply the above algorithm to S_nA. Alternatively, still omniscient, we use the structure of figure 5 (which is simply a convenient renaming of our array-processor, with the first two processors and SSM's combined to form M_0). Using this structure we could attempt to perform Gaussian elimination using the following scheme. The elements of $A^{(0)}$ would enter M_0 and then follow the same route as in the earlier structure. However as the k^{th} column a_{*k} enters and leaves M_0, it would be replaced by $S_k a_{*k}$, as $S_k a_{*k}$ passes through P_j, we would find the j^{th} column permuted by T_k; i.e., M_j would contain S_kT_j and thus be ready for column k, and as column k passed by we would apply τ_{k+1} to the j^{th} column.

Unfortunately, the above method is not feasible, for we do not know the permutation τ_k until all of $a_{*k}^{(k-1)}$ is computed. However a modified version can be used, and we proceed to describe it.

Assume for the moment that $k \geq 4$. As above, the k^{th} column enters and leaves M_0; (as we shall shortly see) at the time it enters M_0, permutation S_{k-3} has been determined, but S_{k-2} and S_{k-1} are not yet known. So we use M_0 to apply the permutation T_{k-3} to $a_{*k}^{(0)}$ obtaining $S_{k-3}a_{*k}^{(0)}$. As before the successive elements of $S_{k-3}a_{*k}^{(0)}$ are sent successively through $P_1,P_2,\ldots,P_{k-3}$. As this occurs, τ_{k-2}, now known, is applied to these columns. The contents of SSM_j, $1 \leq k \leq k-3$, are at this time, as we shall arrange, $S_{k-3}a_{*j}^{(n)}$, $1 \leq k \leq k-3$, respectively. Thus they are in the same order as $S_{k-3}a_{*k}$ and we may calculate $S_{k-3}a_{*k}^{(k-4)}$ and send it directly to P_k (not P_{k-3}); at the same time we may send $S_{k-3}a_{*,k-3}^{(k-3)}$ from P_{k-3} to P_k, where we compute $S_{k-2}a_{*k}^{(k-3)}$, and pass it appropriately permuted into M_k. The important consideration, easily verified by computing times, is that τ_{k-2} is currently available and that, currently, in P_{k-2} we are computing $S_{k-2}a_{*,k-2}^{(k-2)}$ and in P_{k-1} we are computing $S_{k-2}a_{*,k-1}^{(k-2)}$.

Now the following method is employed: As we compute $S_{k-2}a_{*,k-1}^{(k-2)}$ we look for its pivot entry. We do this by holding the $(k-1)^{st}$ entry as it comes by. If a larger entry, say the ℓ^{th} appears, we give what was previously entry $(k-1)$ the index ℓ and store it, replacing the $(k-1)^{st}$ entry by the ℓ^{th}. We say tentatively that $\tau_{k-1}=(k-1,\ell)$. <u>Exactly the same</u> interchanges are performed on the entries entering SSM_{k-2} and SSM_k. Thus, when the k^{th} column is completely loaded into SSM_k, SSM_{k-2} contains $S_{k-1}a_{*,k-2}^{(n)}$, SSM_{k-1} contains $S_{k-1}a_{*,k-1}^{(k-2)}$, SSM_k contains $S_{k-1}a_{*,k}^{(k-3)}$; the pivot entry is known for column $k-1$ and τ_{k-1} is known (to be sent to M_0 and $P_1,P_2,\ldots,P_{k-3}$). We are now ready to compute $S_{k-1}a_{*,k}^{(k-1)}$. This requires two processors and we use P_{k-2} (which is free, for the column that would normally go to that processor is being sent to P_{k+1}) and P_k. $S_{k-1}a_{*,k}^{(k-1)}$ is computed and, as it is stored into M_k, it becomes $S_ka_{*,k-1}^{(k-1)}$. At the same time, the pivot element and τ_k are obtained simultaneously.

Let us summarize the timing aspects of the above by noting at what tick the first element of column k arrives at various places in the array-processor.

Tick	First entry arrives at	Current value of column k
$n(k-1)+1$	M_0	$A_{*k}^{(0)}$
$nk+1$	P_1	$S_{k-3}A_{*k}^{(0)}$
$nk+k-4$	P_{k-4}	$S_{k-3}A_{*,k}^{(k-5)}$
$nk+k-3$	P_k	$S_{k-3}A_{*,k}^{(k-4)}$
$n(k+1)+k-3$	P_{k-2}	$S_{k-1}A_{*,k}^{(k-3)}$
$n(k+1)+k-2$	P_k	$S_{k-1}A_{*,k}^{(k-2)}$
$n(k+2)+k-1$	P_k	$S_{k-1}A_{*,k}^{(k-1)}$
$n(k+2)+k$	P_k	$S_kA_{*,k}^{(k)}$

Finally we note that the above procedure requires the first three columns to be appropriately initialized; we omit the description of this straight-forward procedure.

5. Sparse Matrices

We now indicate how the above methods are used for sparse matrices. We use the following word format:

0	F_1	F_2	F_3	row	column	value

Here the 0 is the leading zero required for valid data, F_1, F_2,F_3 are flag bits. "Row" and "column" are 20 bit fields, allowing for matrices of size $(2^{20}-1)$ by $(2^{20}-1)$ and the value field might be 84 bits for a word length of 128 bits.

We will allow several columns of a matrix to be in one SSM, but will not split a column between two SSM's. We assume the processor has one special register for each column held in its SSM (this can be avoided at the expense of essentially halving the speed of the array-processor). Gaussian elimination proceeds as before, except that as the k^{th} column passes by the processors, several updates of an element may occur (i.e. computations of the form $a_{ij} \leftarrow a_{ij} - a_{ik} \cdot a_{kj}$) for several columns may reside in the corresponding SSM. They will be computed consecutively (for elements with the same row number will be adjacent). This system, while it may operate synchronously at the bit level, should operate asynchronously at the word level. For whenever all updates that can occur in a processor have occurred, the entry should be immediately sent to the next processor, if that processor is not busy. To speed up the procedure even more it would be appropriate to have first-in, first-out buffers between processors (these could be additional SSM's). Then a processor could send an entry on, even if the next processor is busy. Of course when the k^{th} column is being updated, columns $k-3$, $k-2$, $k-1$, k will be kept in distinct SSM's (so that the speed-up trick described in the last section may be used). But as the $(k+1)$st column comes by, the $(k-3)$th column will be moved to the preceding SSM, if there is room. If that occurs each of the columns $k-2$, $k-1$, k will be moved to the preceding SSM.

We note that many variants of the above algorithms are possible, and much further investigation and simulation is necessary to determine which is best.

In addition, it has been tacitly assumed that the main computer, to which our array-processor has been attached, will initially permute the rows and columns of the matrix to make it close to triangular or bounded so as to minimize "fill-in". Standard algorithms for this purpose may also be performed in our array-processor (for example, during partial pivoting it may be better to choose as a pivot element, not the largest, but one which is not too small, but whose row is sparse).

Similar considerations apply to other standard algorithms in matrix theory.

REFERENCES

1. P. Armstrong and M. Rem, A serial sorting machine, Journal of Computers and Electrical Engineering (to appear).

2. I.S. Duff and G. W. Stewart, Sparse matrix proceedings, 1978, SIAM (1979), ISBN 0-89871-160-6.

3. D. Heller, A survey of parallel algorithms in numerical linear algebra, SIAM Review 20 (1978), p. 740-777.

4. A. Jennings, Matrix computation for engineers and scientists, Wiley (1977), ISBN 0-471-99421-9, p. 250-278.

5. A. K. Jones and P. Schwarz, Experience using multiprocessor systems -- A status report, ACM Computing Surveys 12 (1980), p. 121-165.

6. D. Knuth, The art of computer programming V. 3 -- Sorting and searching, Addison-Wesley (1973), ISBN 0-201-03803-X.

7. H. T. Kung and C. Leiserson, Systolic arrays (for VLSI), in [2], p. 256-282.

8. C. Mead and L. Conway, Introduction to VLSI systems, Addison-Wesley (1980), ISBN 0-201-04358-0.

9. W. Orchard-Hays, Advanced linear programming techniques, McGraw-Hill (1968).

10. D. J. Rose and R. A. Willoughby, Sparse matrices and their applications -- proceedings of a symposium, Plenum (1972), ISBN 0-306-30587-9

11. H. S. Stone, Parallel processing with the perfect shuffle, IEEE Trans. on Comp. c-20 (1971), p. 153-161.

12. J. A. Tomlin, Modifying triangular factors of the basis in the simplex method, in [10], p. 77-85.

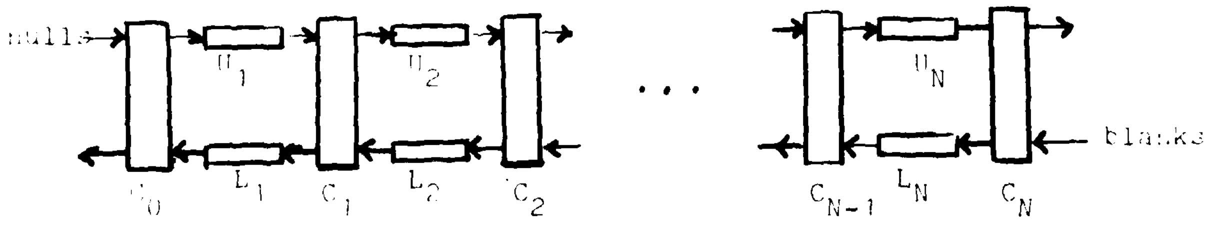

Figure 1

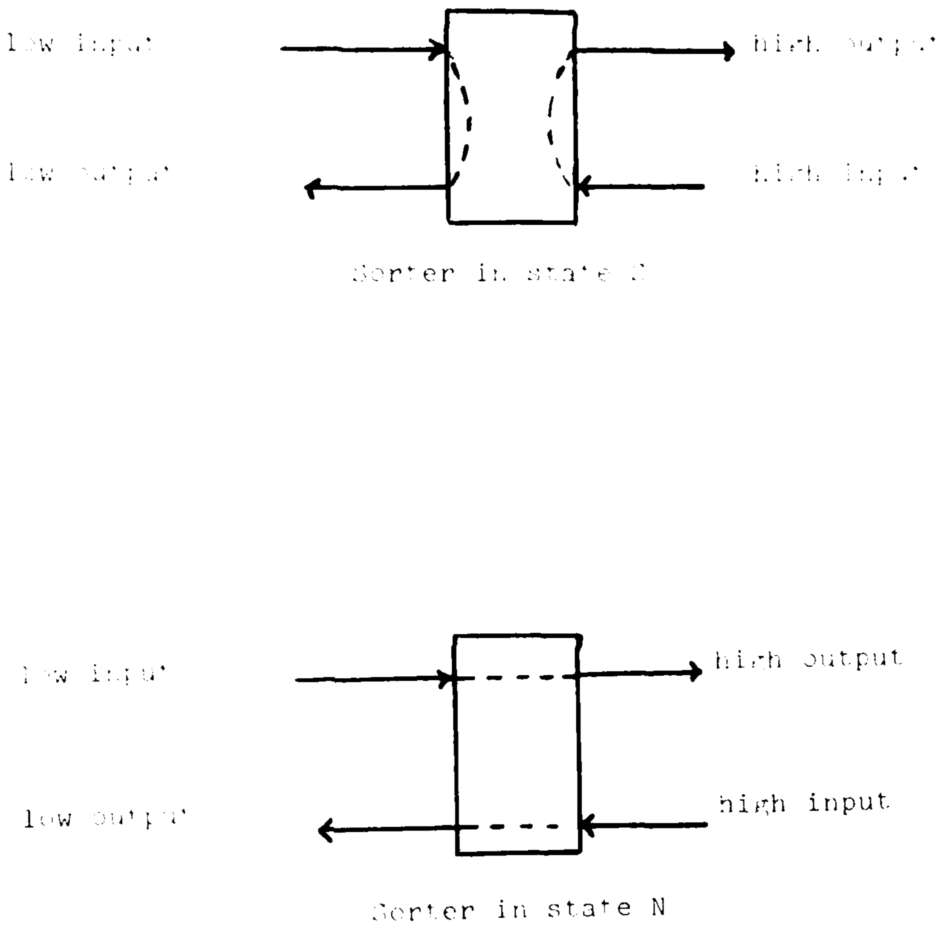
low input
high output
low output
high input
Sorter in state C
low input
high output
low output
high input
Sorter in state N

Figure 2

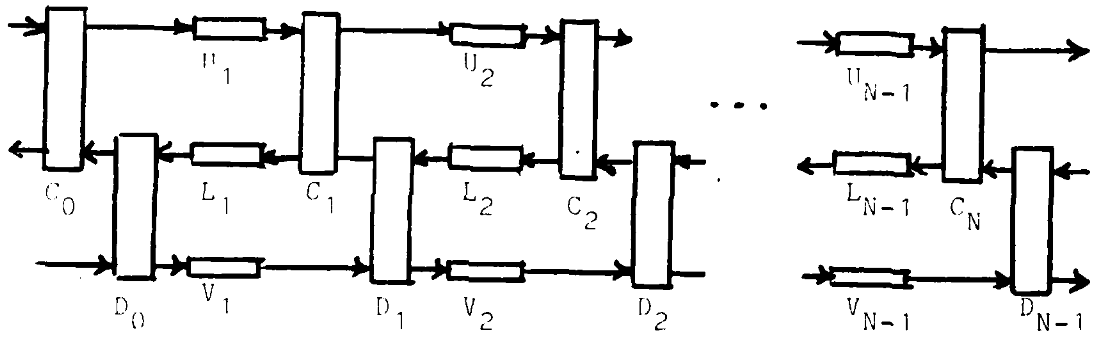

Figure 3

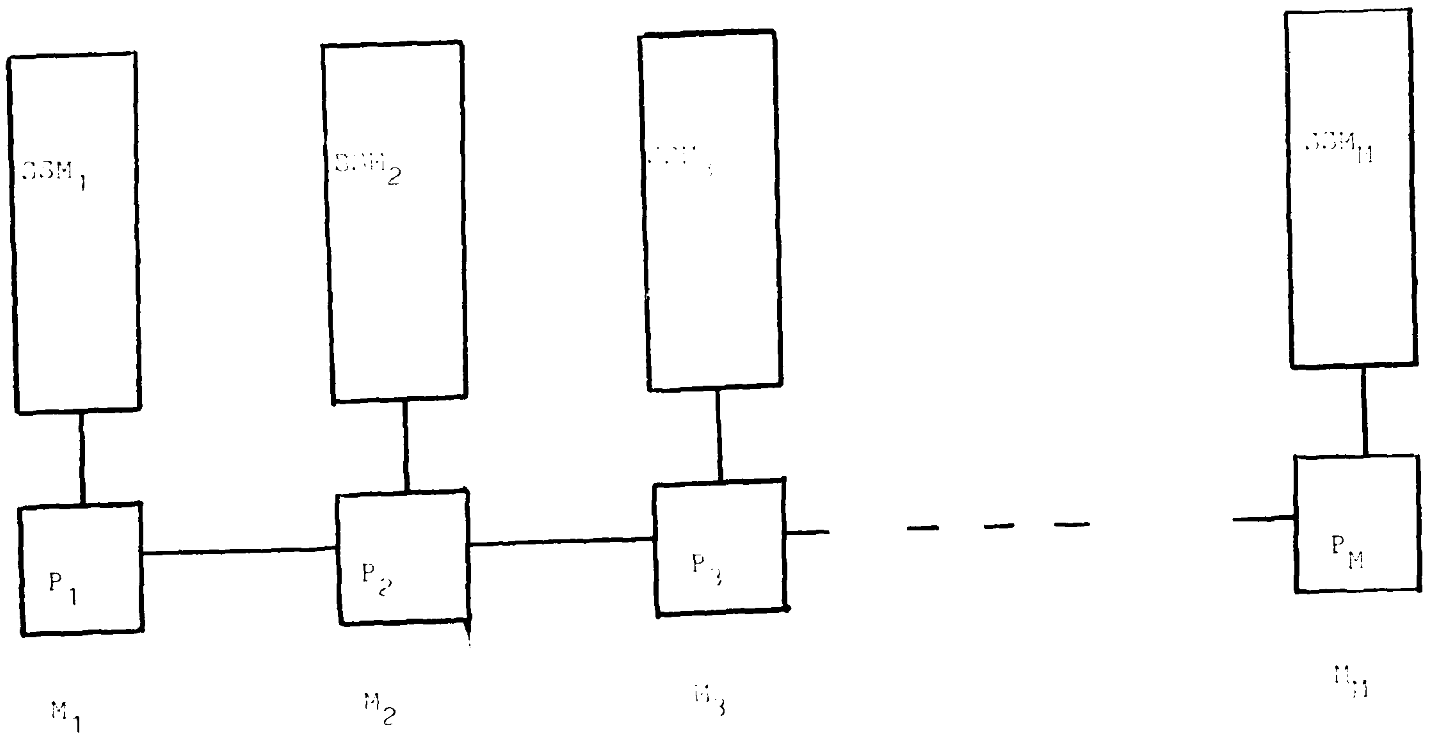

Figure 4

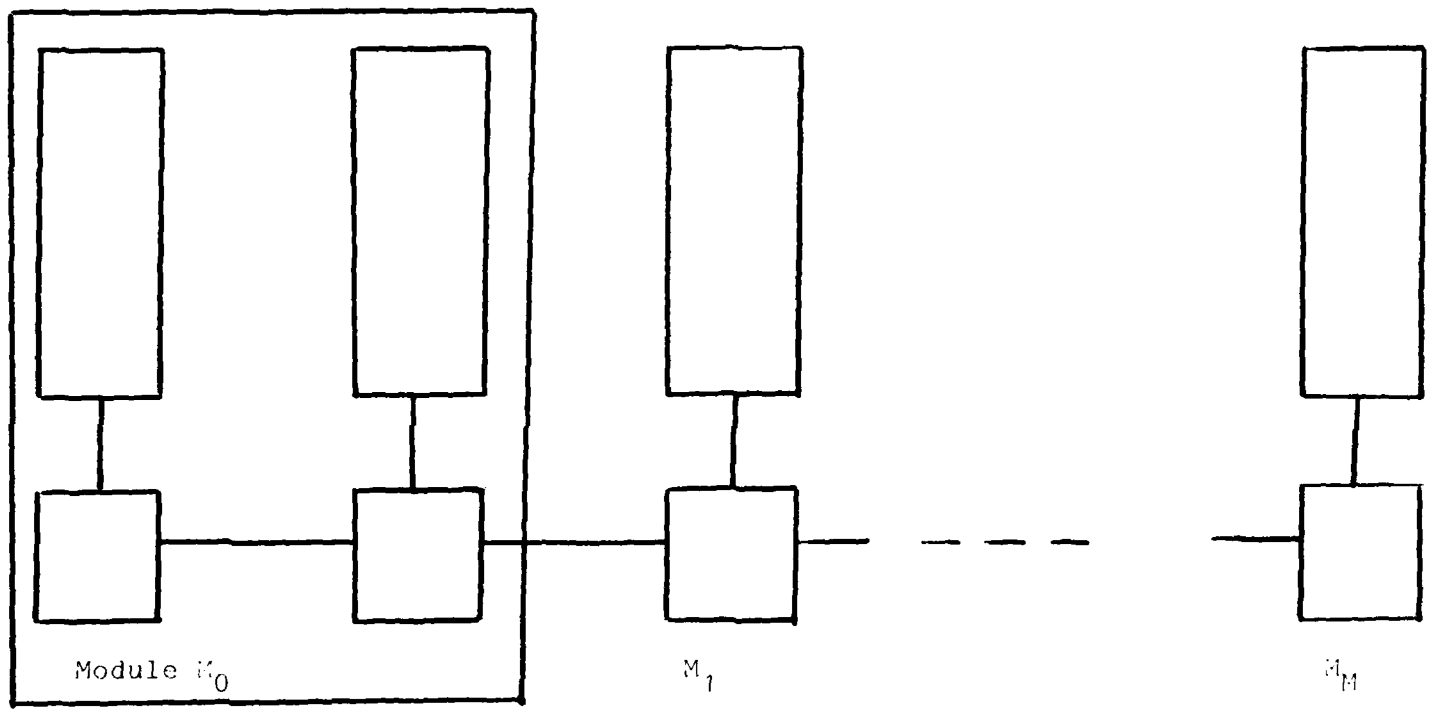

Figure 5